Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 317 (2015) 47–54

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Transformation of a PID Controller for Numerical Accuracy[†]

N. Damouche[1,*] M. Martel[2,*] A. Chapoutot[3,**]

[*] *Université de Perpignan Via Domitia, DALI, Perpignan, France*
[*] *Université Montpellier & CNRS, LIRMM, UMR 5506, Montpellier, France*
[**] *ENSTA ParisTech, Palaiseau, France*

Abstract

Numerical programs performing floating-point computations are very sensitive to the way formulas are written. Several techniques have been proposed concerning the transformation of expressions in order to improve their accuracy and now we aim at going a step further by automatically transforming larger pieces of code containing several assignments and control structures. This article presents a case study in this direction. We consider a `PID` controller and we transform its code in order to improve its accuracy. The experimental data obtained when we compare the different versions of the code (which are mathematically equivalent) show that those transformations have a significant impact on the accuracy of the computations.

Keywords: Numerical Accuracy, Semantics-Based Program Transformation, Floating-Point Arithmetic, Validation of Numerical Programs.

1 Introduction

Numerical programs performing floating-point computations are very sensitive to the way formulas are written. Indeed, small syntactic changes in the arithmetic expressions which do not modify their mathematical meaning may lead to significant changes in the result of their evaluation. This sensitivity to the way expressions are written is due to the particularities of the floating-point arithmetic in which, for example, addition is not associative or multiplication is not invertible [1,11,12]. In addition, it is very difficult to guess which writing of a formula gives the best accuracy when evaluated with floating-point numbers. These last years, abstract interpretation techniques [3,5] have been developed to infer safe approximations of the round-off error on the result of a computation [2,6,7].

[1] Email: nasrine.damouche@univ-perp.fr
[2] Email: matthieu.martel@univ-perp.fr
[3] Email: alexandre.chapoutot@ensta-paristech.fr
[†] This work was supported by the ANR Project ANR-12-INSE-0007 "CAFEIN".

http://dx.doi.org/10.1016/j.entcs.2015.10.006

Our work concerns the automatic transformation of floating-point computations in order to improve their numerical accuracy. Several results have been obtained concerning the transformation of expressions [9,10] and now we aim at going a step further by automatically transforming larger pieces of code containing several assignments and control structures.

This article presents a case study in this direction. We consider a `PID` controller and we transform its code in order to improve its accuracy. More precisely, we take an initial `PID` code and we apply to it several processings in order to generate other `PID` programs which are mathematically equivalent to the initial one but more accurate in computing. The first transformation only rewrites the assignments while, in the second transformation, the loop is unfolded. While these transformations are made by hand, they are applied systematically, in a way which we aim at automatizing in future work. The experimental data obtained when we compare the executions of the three codes (which are mathematically equivalent) show that those rewritings have a significant impact on the accuracy of the computations.

The rest of this article is organized as follows. Section 2 introduces the original controller $\texttt{PID}_1$. The transformations are done in Sections 3 and 4, yielding $\texttt{PID}_2$ and $\texttt{PID}_3$. The experimental results are presented in Section 5 and Section 6 concludes.

2 Description of the PID Controller

In this section, we give a brief description of the original `PID` program of Listing 1. This kind of algorithm is used in embedded and critical systems to maintain a measure m at a certain value named *setpoint* c. The error being the difference between the setpoint and the measure, the controller computes a correction based on the integral i and derivative d of the error and also from a proportional error term p. A weighted sum of these terms is computed. The weights are used to improve the reactivity, the robustness and the speed of the program. The three terms are:

i) The proportional term p: the error e is multiplied by a proportional factor k_p,

$$p = k_p \times e \ .$$

ii) The integral term i: the error e is integrated and multiplied by an integral factor k_i,

$$i = i + (k_i \times e \times dt) \ .$$

iii) The derivative term d: The error e is differentiated with respect to time and is then multiplied by the derivative factor k_d. Let $eold$ denote the value of the error at the previous iteration, we have:

$$d = kd \times (e - eold) \times \frac{1}{dt} \ .$$

In practice, there exists many other ways to compute the terms d and i. In our implementation they are computed by Euler's method and the rectangle method

Listing 1: Source code of $\texttt{PID}_1$.

```
kp = 9.4514;  ki = 0.69006;  kd = 2.8454;  invdt =  5.0;  dt = 0.2;
m = 8.0;      c = 5.0;       eold = 0.0;
while true do
     e = c - m;
     p = kp * e;
     i = i + ki * dt * e;
     d = kd * invdt * (e - eold);
     r = p + i + d;
     eold = e;           /* updating memory   */
     m = m + 0.01 * r;   /* computing measure: the plant */
```

respectively. The transformations described in the next sections are independent of these specific algorithms.

3 How to Get a More Accurate PID?

In this section, we detail the different steps needed to transform the original PID, named $\texttt{PID}_1$, into a new equivalent program named $\texttt{PID}_2$. The main idea consists of developing and simplifying the expressions of $\texttt{PID}_1$ and inlining them within the loop, in order to extract the constant expressions and to reduce the number of operations. At the n^{th} iteration of the loop, we have:

$$\begin{aligned} p_n &= k_p \times e_n \ , \\ i_n &= i_{n-1} + k_i \times e_n \times dt \ , \\ d_n &= k_d \times (e_n - e_{n-1}) \times \frac{1}{dt} \ . \end{aligned}$$

If we inline the expressions of p_n, i_n and d_n in the formula of the result expression r_n and after extracting the common factors, we get:

$$r_n = e_n \times \left(k_p + k_d \times \frac{1}{dt}\right) + i_0 + k_i \times dt \times \sum_{i=1}^{n} e_i - k_d \times e_{n-1} \times \frac{1}{dt} \ . \qquad (1)$$

Then we remark that there exists some constant sub-expressions in Equation (1). So, we compute them once before entering into the loop. We have:

$$c_1 = k_p + k_d \times \frac{1}{dt} \ , \qquad c_2 = k_i \times dt \ , \qquad c_3 = k_d \times \frac{1}{dt} \ .$$

Next, we record in a variable s the sum $s = \sum_{i=0}^{n-1} e_i$ which adds the different errors from e_0 to e_{n-1}. Finally, we have:

$$r_n = R + i_n \qquad \text{with} \qquad R = c_1 \times e_n - c_3 \times e_{n-1} \ .$$

Our $\texttt{PID}_2$ algorithm is given in Listing 2.

Listing 2: Source code of $\texttt{PID}_2$.

```
kp = 9.4514;  ki = 0.69006;  kd = 2.8454;  invdt =  5.0;  dt = 0.2;
m = 8.0;      c = 5.0;       eold = 0.0;   R = 0.0;       s = 0.0;
c1 = kp + kd * invdt;  c2 = kd * invdt;  c3 = ki * dt;
while true do
      e = c - m;
      i = i + c2 * e;
      R = (c1 * e) - (c3 * eold);
      r = R + i;
      eold = e;
      m = m + 0.01 * r;
```

4 How to Get an Even More Accurate PID?

The initial PID can be transformed even more drastically by unfolding the loop. In our case, we arbitrarily choose to unfold it four times in order to keep for each execution the sum of the last four errors. Then, we change the order of the operations, either by summing the terms pairwise, or in increasing or decreasing order. The process applied to get the third PID algorithm, named $\texttt{PID}_3$, is given in the following. Let us start by unfolding four times the integral term, as usually done by static analyzers or compilers:

$$i_{n-1} = i_{n-2} + k_i \times dt \times e_{n-1} \qquad i_{n-2} = i_{n-3} + k_i \times dt \times e_{n-2}$$
$$i_{n-3} = i_{n-4} + k_i \times dt \times e_{n-3} \qquad i_{n-4} = i_{n-5} + k_i \times dt \times e_{n-4}$$

We inline the previous expressions in i_n. We obtain:

$$i_n = i_{n-5} + (k_i \times dt \times e_{n-4}) + (k_i \times dt \times e_{n-3}) + (k_i \times dt \times e_{n-2}) + (k_i \times dt \times e_{n-1}) + (k_i \times dt \times e_n) \quad (2)$$

with $i_{n-5} = i_0 + k_i \times dt \times \sum_{i=1}^{n-5} e_i$. Equation (2) can be even more simplified, we have:

$$i_n = i_0 + k_i \times dt \times \sum_{i=1}^{n-5} e_i + (k_i \times dt \times ((((e_{n-4} + e_{n-3}) + e_{n-2}) + e_{n-1}) + e_n)) \ .$$

Now, if we come back to the result expression after having done some manipulations, like developing the derivative and factorizing, we obtain as final expression:

$$r_n = e_n \times \left(k_p + k_d \times \frac{1}{dt}\right) + i_0 + k_i \times dt \times \sum_{i=1}^{n-5} e_i - k_d \times \frac{1}{dt} \times e_{n-1} + k_i \times dt \times ((((e_{n-4} + e_{n-3}) + e_{n-2}) + e_{n-1}) + e_n) \ .$$

Denoting by $s = \big(((e_{n-4} + e_{n-3}) + e_{n-2}) + e_{n-1}\big)$, $k_1 = k_p + \big(k_d \times \frac{1}{dt}\big)$, $k_2 = k_i \times dt$

and $k_3 = k_d \times \frac{1}{dt}$, the final expression of r_n is:

$$r_n = R + i + k_2 \times \left(s + \sum_{i=1}^{n-5} e_i\right) \qquad \text{with} \qquad R = (e_n \times k_1) - (k_3 \times e_{n-1}) \ .$$

The complete code of $\texttt{PID}_3$ is given in Listing 3. Remark that, the transformation proposed in this section leads to a larger program requiring more memory for its execution. While it allows more transformations it may be irrelevant in some contexts where memory is critical.

5 Experimental Results

Let us focus now on the execution of the three PID programs. In our Python implementation using the *GMPY2* library for multiple precision computations, our results show that there is a significant difference between $\texttt{PID}_1$, $\texttt{PID}_2$ and $\texttt{PID}_3$ as soon as on the second or third digit of the decimal values of the result. To better visualize these results, the curves corresponding to the three PID algorithms are given in Figure 1. We can observe a significant difference between the curves corresponding to the three PID, mainly between 0 and 150 of the x-axis.

Figure 2 shows the difference between $\texttt{PID}_1$ and $\texttt{PID}_3$. This difference, which is important, is computed with many precisions. So, the same behavior was observed by using 24, 53 and 50000 bits of the mantissa. The error between $\texttt{PID}_1$ and $\texttt{PID}_3$ oscillates between -0.1 and 0.25 while the value ranges between 5 and 8.

We also observe that the differences between $\texttt{PID}_1$ and $\texttt{PID}_2$ are negligible. Concerning $\texttt{PID}_1$ and $\texttt{PID}_3$, we can remark that a small syntactic change in the code indeed yields an important difference in term of accuracy. For example, let us take the following expression r of $\texttt{PID}_1$ and let us just inline the three terms p, i and d and factorize e in it. Initially, $r = p + i + d$ and we obtain after factorizing:

$$r' = e \times \left(k_p + k_i \times dt + k_d \times \frac{1}{dt}\right) + i_0 - \left(k_d \times \frac{1}{dt} \times eold\right) \ .$$

With this simple modification, the difference in accuracy is already important, as shown in Figure 3 which gives the difference between r and r' and between m and m' for the first iterations of the loop.

Listing 3: Source code of $\texttt{PID}_3$.

```
kp = 9.4514;  ki = 0.69006;  kd = 2.8454;  invdt =  5.0;  dt = 0.2;
R = 0.0;  S = 0.0;  s = 0.0;  m = 8.0;  c = 5.0;  eold= 0.0;
e1 = e2 = e3 = e4 = 0.0; k1 = kp + kd * invdt; k2 = ki * dt; k3 = kd * invdt;
while true do
     e = c - m;
     i = i + k2 * e;
     R = (k1 * e) - (k3 * eold);
     S = s + (e4 + (e3 + (e2 + e1)));
     r = R + i + (k2 * S);
     eold= e;  e4 = e3;  e3 = e2;  e2 = e1;  e1 = e;
     m = m + 0.01 * r;
```

6 Conclusion

In this article, our attention focused on the transformation of a standard PID Controller. We believe that it is possible to obtain automatically the entire transformation going from $\texttt{PID}_1$ to $\texttt{PID}_2$ and to $\texttt{PID}_3$ (see sections 3 and 4) by using systematic and general rules independent of the sample program used in this case study. These rules include the inlining of expressions, partial evaluation and loop unfolding. The results obtained when running the three codes show that these transformations impact significantly the accuracy of the results (several percents).

Currently, we are developing a software, based on the rules mentioned in the former paragraph as well as rewriting rules for the expressions (associativity, commutativity, etc.) This tool aims at taking as input an initial program, like $\texttt{PID}_1$, and at generating automatically other PID programs that are equivalents mathematically and more precise.

A key issue concerns the combinatory explosion which occurs during the transformation process since many rules my be used at each step. A full exploration of the set of equivalent programs is not realistic and we aim at developing more tractable techniques using abstractions.

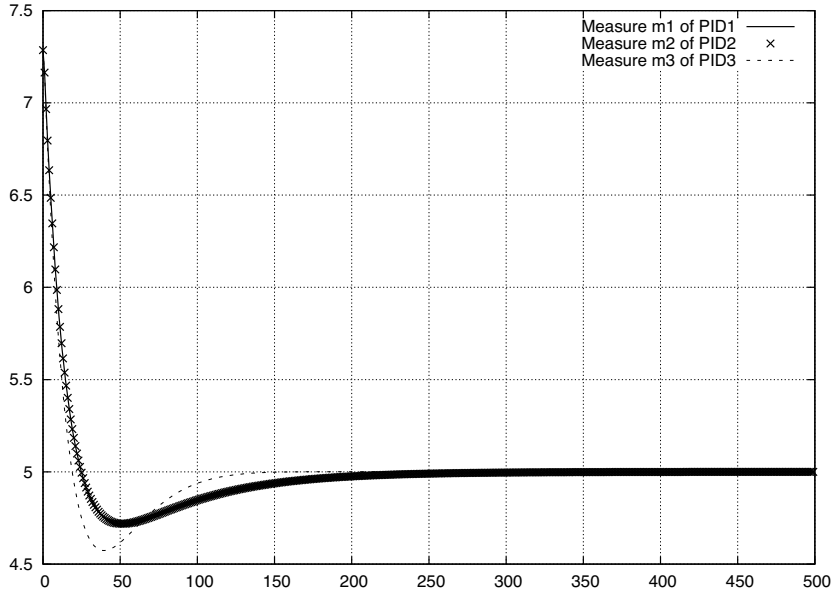

Fig. 1. Value of the measure m in the three PID algorithms.

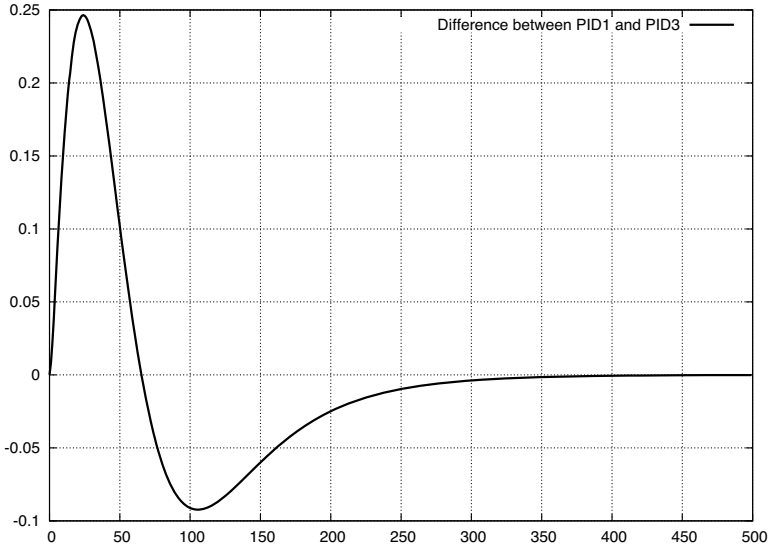

Fig. 2. Difference between the values of the measure m in $\texttt{PID}_1$ and $\texttt{PID}_3$.

It	r	r'	m	m'
1	−71.449234	−71.449242	7.285508	7.285508
2	−12.165627	−12.993700	7.163851	7.155571
3	−19.748718	−21.010429	6.966364	6.945466
4	−17.074722	−18.747597	6.795617	6.757990
5	−16.089169	−18.077232	6.634725	6.577218
6	−14.934349	−16.752943	6.485382	6.409688
7	−13.892138	−15.672437	6.346460	6.252964
8	−12.913241	−14.609431	6.217328	6.106869
9	−12.000034	−13.609982	6.097328	5.970769
10	−11.147227	−12.662717	5.985856	5.844142

Fig. 3. Comparaison between results r and r' and between the corresponding measures m and m'.

References

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-2008 edition, 2008.

[2] Alexandre Chapoutot. Interval slopes as a numerical abstract domain for floating-point variables. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337, pages 184–200. Springer, 2010.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, New York, NY, 1977.

[4] P. Cousout and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Principles of Programming Languages*, pages 178–190. ACM, 2002.

[5] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and V. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.

[6] Eric Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *Static Analysis - 20th International Symposium, SAS 2013. Proceedings*, volume 7935, pages 1–3. Springer, 2013.

[7] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011. Proceedings*, volume 6538, pages 232–247. Springer, 2011.

[8] Arnault Ioualalen and Matthieu Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *Static Analysis Symposium (SAS'12)*, volume 7460 of *Lecture Notes in computer Science*, pages 75–93. Springer Verlag, 2012.

[9] Matthieu Martel. Semantics-based transformation of arithmetic expressions. In *Static Analysis, 14th International Symposium, SAS 2007, Proceedings*, volume 4634, pages 298–314. Springer, 2007.

[10] Matthieu Martel. Accurate evaluation of arithmetic expressions (invited talk). *Electronic Notes Theoretical Computer Science*, 287:3–16, 2012.

[11] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[12] Michael L. Overton. *Numerical computing with IEEE floating point arithmetic - including one theorem, one rule of thumb, and one hundred and one exercices*. SIAM, 2001.